Substring Parsing for Arbitrary Context-Free Grammars

Jan Rekers
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: rekers@cwi.nl

Wilco Koorn
Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

Abstract

A substring recognizer for a language L determines whether a string s is a substring of a sentence in L, i.e., *substring-recognize*(s) succeeds if and only if $\exists v, w\colon vsw \in L$. The algorithm for substring recognition presented here accepts general context-free grammars and uses the same parse tables as the parsing algorithm from which it was derived. Substring recognition is useful for *noncorrecting* syntax error recovery and for incremental parsing. By extending the substring *recognizer* with the ability to generate trees for the possible contextual completions of the substring, we obtain a substring *parser*, which can be used in a syntax-directed editor to complete fragments of sentences.

1 Introduction

A recognizer for a language L determines whether a sentence s belongs to L. A substring recognizer performs a more complicated job, as it determines whether s can be *part* of a sentence of L.

A recently developed substring recognition algorithm [4] uses an ordinary LR parsing algorithm with special parse tables. For ordinary parsing, this parsing algorithm is limited to LR(1) grammars, but the more complicated nature of substring recognition limits it to bounded-context grammars (see Section 3).

In Section 4 we describe a substring recognition algorithm that does not suffer from this drawback. It accepts general context-free grammars and uses the same parse tables as the ordinary parser. Our algorithm is based on the pseudo-parallel parsing algorithm of Tomita [17], which runs a dynamically varying number of LR parsers in parallel and accepts general context-free grammars. In Section 5 we extend the substring *recognizer* into a substring *parser* that generates trees for the possible completions of the substring.

Partial support received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II - GIPE II) and from the Netherlands Organization for Scientific Research – NWO, project *Incremental Program Generators.*

2 Applications

2.1 Syntax error recovery

In its simplest form, a parser stops at the first syntax error found. If it has to find as many errors in the input as possible, it can try to correct the error in order to continue parsing. Spurious errors are easily introduced, however, if the parser makes false assumptions about the kind of error encountered.

Substring parsing can be used to implement *non*correcting syntax error recovery. If an ordinary parser detects a syntax error on some symbol, the substring parser can be started on the next symbol to discover additional syntax errors. Using this method, it is not necessary to let the parser make any assumption about how to correct the error, or to let it skip input until a trusted symbol is found.

Richter defines noncorrecting syntax error recovery with the aid of substring parsing and interval analysis in a formal framework [15]. He proves that his technique does not generate spurious errors, but is not explicit about its implementation.

He notes, however, that there are difficulties in keeping the substring parser deterministic due to a limitation on the class of grammars accepted. Our technique could be useful here, as it implements the required substring analysis for general context-free grammars.

2.2 Completion tool

In Section 5 we will show how the substring recognizer can be extended so that it generates parse trees for the possible completions of a substring. As the total number of possible completions will often be infinite, only generic completions are generated. A syntax-directed editor could use these to complete fragments of sentences in accordance with the grammar used, or to guess the continuation of what the user is typing.

2.3 Incremental parsing

Another application for substring parsing is in incremental parsing. Incremental parsing can be performed by attaching parser states to tokens [3, 1, 18]. After a modification has been made, the parser is restarted in a saved state, at a point in the text just before the modification. Parsing stops when the parser reaches a token after the modification in an old configuration (if ever). These methods are very good as to minimizing the amount of recomputation after a modification, but require a huge amount of memory for storing the states of the parser (parse stacks with partial parse trees as elements).

Ghezzi and Mandrioli present an alternative technique for incremental parsing. [7, 8] If the string $x\bar{x}z\bar{y}y$ is modified to $x\bar{x}\hat{z}\bar{y}y$, where $\bar{x}$ and $\bar{y}$ have length k, with k the look-ahead used by the parser, then the parse trees previously generated for x and y are still valid after the modification. All subtrees previously generated for x and y can thus be abbreviated by their top nonterminals, which minimizes the length of the string to be reparsed. This technique is both time and space efficient, but is not applicable to general context-free parsing as it requires a fixed look-ahead. In our particular case, we need incremental parsing in a syntax-directed editor that uses the Tomita parser. By running a varying number of LR-parsers in parallel, the Tomita parser adjusts its look-ahead dynamically to the amount needed, and is thus not limited to an a priori known k.

Incremental parsing can also be achieved in another manner: after a modification has been made in the text, find the substring s' belonging to the smallest subtree that contains the modification in the stored parse tree. If the type of this subtree is T and s' can be parsed as a tree of type T, replace the old subtree by the new one. If s' fails to parse, it may be the case that the modification introduced a syntax error, or that the subtree has been chosen too small. These two cases must be distinguished, as the incremental parser proceeds in a different way in each case. A substring parser can provide a hint as to which of the two possibilities is actually the case. If the substring parser fails on s', the modification will be syntactically incorrect in any context, and an error message can be given. If the substring parser succeeds, a larger subtree is chosen and parsing is retried. This can be more time consuming than remembering parser states, but the amount of memory needed is far less. We consider using this scheme in the syntax-directed editor GSE [11], but it has to be investigated further as a lot of work is still performed twice.

3 Related work

Cormack [4] describes a substring parse technique for Floyd's class of bounded context or BC(1,1) grammars [6], and implements the substring parser Richter mentions [15]. A grammar is BC(1,1) if for every rule $A ::= \alpha$, if some sentential form contains $a\alpha b$ where α is derived from A then α is derived from A in *all* sentential forms containing $a\alpha b$. This class is smaller than LR(1). The solution of Cormack consists in using an ordinary LR automaton, but a special parse table constructor. The sets of items generated do not only contain items of the form $A ::= \alpha{\bullet}\beta$ but also "suffix items" of the form $A ::= \cdots{\bullet}\beta$. These suffix items denote partial handles whose origins occur before the beginning of the input. The generated parse tables are deterministic, provided that the grammar is BC(1,1). This substring parser is used for noncorrecting error recovery in a parser for Pascal. The BC(1,1) limitation on the grammar caused problems in the definition of Pascal, which where alleviated by permitting the parse table generator to rewrite the grammar if necessary.

Lang describes a method for parsing sentences containing an arbitrary number of unknown parts of unknown length [12]. The parser produces a finite representation of all possible parses (often infinite in number) that could account for the missing parts. The implementation of this method is based on Earley parsing [5], as is the Tomita algorithm we use in our own substring parser. The basic idea of Lang's method is that "in the precence of the unknown subsequence $*$, scanning transitions may be applied any number of times to the same computation thread, without shifting the input stream." This process terminates, as parsers in the same state are joined and the number of states is finite. This method is very elegant and powerful, and can be used as a substring parser (by providing it with the string "$*s*$"). We will not use it, however, as it is more general than what we need. Whether it would be efficient enough for interactive purposes is unclear.

Snelting presents a technique to complete the right-hand side of unfinished sentences [16] (also see Section 5.2).

4 Substring Recognition

4.1 Tomita parsing

We base the implementation of our substring parser on Tomita's algorithm. This algorithm runs several simple LR parsers in parallel. It starts as a single LR parser, but, if it encounters a conflict in the parse table, it splits in as many parsers as there are conflicting possibilities. These independently running simple parsers are fully determined by their parse stack. When two parsers have the same state on top of their stack, they are joined in a single parser with a forked stack. A reduce action which goes back over a fork in a parse stack, splits the corresponding parser again into two separate parsers. If a parser hits an error entry in the parse table, it is killed by removing it from the set of active parsers. The possibility to run several parsers in parallel makes the Tomita algorithm very well suited for substring parsing.

For a full description of the Tomita parsing algorithm we refer to Tomita [17], to Nozohoor-Farshi who corrected an error in the algorithm concerning ϵ-productions [13], or to Rekers who extended the algorithm to the full class of context-free grammars by including cyclic grammars[1] [14]. For a detailed explanation of LR parsing [2, ch. 4.7] is recommended.

[1]Grammars in which $A \overset{+}{\Longrightarrow} A$ is a possible derivation

4.2 The grammar

The grammar for which our substring recognition algorithm works should be reduced in such a way that it does not contain non-terminals that cannot produce any terminal string or ϵ. These non-terminals can be identified easily, and all rules in which they appear should be removed from the grammar. This clean-up operation does not affect the language recognized. [9, p. 73-76]

Useless symbols and unreachable rules do not influence substring parsing as these are ignored by the parse table generator. This is due to the fact that LR parse tables are generated top-down, starting with the start symbol of the grammar, and that useless symbols and unreachable rules are, by definition, unreachable from the start symbol.

4.3 The algorithm

If we have to determine whether a string $s_0 \cdots s_n$ is a substring of a sentence in a language L, we start the substring recognition process by generating, for each state directly reachable under s_0, a parser with this state on its stack. These parsers will process $s_1 \cdots s_n$.

We will show how an individual parser processes an action, but we will not discuss the management of the different parsers, as this is done in the same way as in ordinary Tomita parsing. The parser obtains an action from the parse table with the state on top of its stack and with input symbol s_k. This can be a *shift*, *error* or *reduce*-action, and is processed in the following manner:

- A (shift *state'*)-action is processed as in normal parsing: *state'* is pushed on the stack and the parser is ready to process s_{k+1}.
- An (error)-action removes the parser from the set of active parsers.
- A (reduce $A ::= \alpha\beta$)-action is processed as follows:
  - If there are at least $|\alpha\beta| + 1$ entries on the parse stack the reduce action is performed as in normal parsing: $|\alpha\beta|$ entries are popped off the stack, and the parse table is consulted, with the state remaining on top of the stack and A, to obtain a state to push on the stack again. The parser is now ready to continue the processing of s_k.

- If there are only $|\beta|$ entries on the stack, only β has been recognized of $A ::= \alpha\beta$; α lies before s_0 and should produce (a part of) a prefix of s_0. This is possible, as all non-terminals in α can produce some terminal string, and all terminals in α trivially do. So the reduction $A ::= \alpha\beta$ may be performed. The states which can be reached directly by a transition under A are the states where parsing may continue. For each of these valid states a new parser is started with that state on the stack. These parsers all proceed to process s_k.
- If there are exactly $|\alpha\beta|$ entries on the stack, $s_0 \cdots s_{k-1}$ reduces to $\alpha\beta$, but the context in which A is to be used is unknown. This is handled in the same way as the previous case.

If there are no parsers left alive after the processing of s_n, the substring parser fails. If there are parsers left, these are currently recognizing rules $A ::= \alpha\beta$, of which (a part of) α has been recognized. As every β can produce some terminal string, these rules can all be finished. This means that the substring parser succeeds if there are parsers remaining after the processing of s_n.

4.4 The parse table generator

The substring parser is controlled by the same parse table as our ordinary parser. To generate this parse table we use an extended version of the lazy and incremental parser generator IPG [10]. The extension concerns the need of the substring parser to know all states which can be reached by a transition under a given symbol. This function needs global information about the parse table, which means that the whole parse table must be known. As a consequence, the lazy aspect of IPG cannot be exploited here and the parse table is always fully expanded. The expanded parse table can also be used by the ordinary parser, of course.

5 Substring Parsing

We extend the substring recognizer into a substring parser by generating parse trees for substrings. The possible parse trees for a substring s are the parse trees of all sentences vsw for which $vsw \in L$ holds. To limit the number of completions we allow v and w to consist both of *terminals* and *non-terminals*, and we generate a parse tree,

START ::= *Stat*
START ::= *Exp*
Stat ::= if *Exp* then *Stat*
Stat ::= if *Exp* then *Stat* else *Stat*
Stat ::= *Id* := *Exp*
Exp ::= *Id*
Exp ::= *Int*
Exp ::= *Exp* + *Exp*
Exp ::= *Exp* $*$ *Exp*
Exp ::= (*Exp*)

Figure 1: A grammar

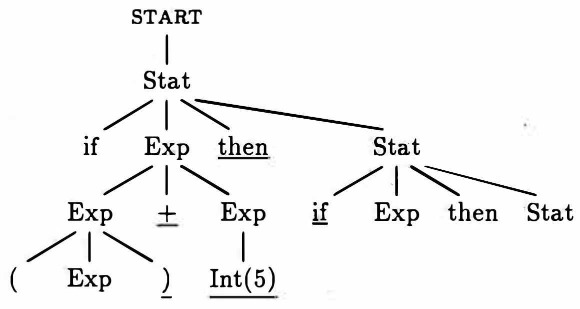

Figure 2: A completion of ") + 5 then if"

corresponding to a sentential form $\sigma_1 s \sigma_2$, only when the frontier of each of its *subtrees* contains at least one symbol of s; i.e., we do not generate subtrees whose frontier lies entirely within σ_1 or σ_2. The trees that we generate are the most general trees, as it is not possible to replace any of their subtrees by a non-terminal such that the frontier still contains s as a substring. Even so, the number of completions can still be infinite. In Section 5.2 we will discuss how to limit this number still further.

For the grammar of Figure 1 and the string ") + 5 then if", a possible completion is the sentential form

$$\underbrace{\texttt{if (} Exp}_{\sigma_1}\ \underbrace{\texttt{) + 5 then if}}_{s}\ \underbrace{Exp\ \texttt{then}\ Stat}_{\sigma_2}$$

whose parse tree is given in Figure 2. To distinguish the leaves of s from those of σ_1 and σ_2, the former are underlined.

5.1 Generating the completions of a substring

LR parsers generate parts of parse trees during a reduction step. On reducing $A ::= \alpha$, the parse stack contains the subtrees created for α. These are assembled in a new node of type A and the subtree created in this way is pushed on the stack. In the substring parser ordinary reductions are treated in the same way.

If the rule $A ::= \alpha\beta$ is reduced with only nodes for β on the stack, however, additional nodes are created for α. In this way, the parse trees for the possible prefixes of s are created.

Parse trees for postfixes of s are created in the same way: after processing s the parser has to finish all rules which are in the process of being recognized. These are the rules in the kernel of the current state of the parser. If only α has been seen from a rule $A ::= \alpha\beta$, the rule is reduced and additional nodes are created for β. It can even be the case that only β has been recognized from a rule $A ::= \alpha\beta\gamma$, and that nodes must be created for both α and γ.

5.2 Further reduction of the number of possible completions

By producing only parse trees that are most general, the number of possible completions is reduced, but it is often still too large and not even always finite. We propose the following rules to limit this number still further:

1. The parse trees generated are kept as compact as possible by disallowing derivations of the form $A \overset{+}{\Longrightarrow} \alpha A$, $A \overset{+}{\Longrightarrow} \alpha A \beta$, and $A \overset{+}{\Longrightarrow} A\beta$, where only A has actually been recognized and all elements of α and β would produce elements in σ_1 or σ_2. Clearly, such derivations can be repeated infinitely often. They are undesirable as they only enlarge σ_1 or σ_2.

   For example, the substring "`) + 5 then if`" also has a possible completion

   $$\underbrace{\texttt{if}\ \textit{Exp}\ \texttt{+ (}\ \textit{Exp}}_{\sigma_1}\ \underbrace{\texttt{) + 5 then if}}_{s}\ \underbrace{\textit{Exp}\ \texttt{then}\ \textit{Stat}}_{\sigma_2}$$

   whose parse tree is given in Figure 3. In this tree a subtree for the rule *Exp* ::= *Exp* + *Exp* has been inserted in the prefix.

2. The number of possible sentential forms for which parse trees are generated is now finite, but these can still have infinitely many parse trees as the grammar may be cyclic. Rekers describes how to parse and generate *parse graphs* for cyclic grammars [14]. The cycles generated in this graph can be removed by his routine *remove-cycles*. This results in a finite number of most general completions.

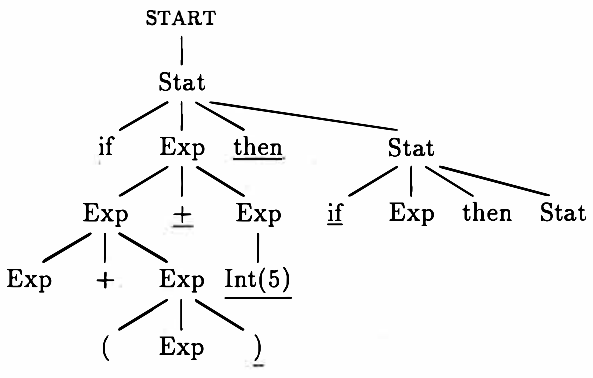

Figure 3: Another possible completion of "`) + 5 then if`"

3. In the generation of the postfixes of s a choice can be made for the "simplest" completion. That is, if a substring can be completed according to both $A ::= \alpha\beta$ and $A ::= \alpha\gamma$, and $|\beta| < |\gamma|$, we prefer $A ::= \alpha\beta$. In the example of Figure 2 this rule forbids the choice of the "if-then-else" rule, as the "if-then" rule already applies. Snelting's rule "*prefer reduce items over shift items*" [16] is similar to ours. It can also be formulated as: if completion according to both $A ::= \alpha$ and $B ::= \alpha\gamma$ ($\gamma \neq \epsilon$) is possible, then prefer $A ::= \alpha$. We consider our rule more appropriate, as we take the case of β being non-empty but shorter than γ into account as well, and we only make the choice if the two rules reduce to the same non-terminal. Otherwise, the rule $A ::= \alpha$ might be preferred over $B ::= \alpha\gamma$, whereas the environment in which the substring is completed needs a tree of type B.

6 Measurements

Our first measurement compares the substring recognizer with the Tomita recognizer from which it was derived to learn the additional costs of substring parsing.[1]

We have taken a grammar of about twenty rules and sentences of increasing length. These were parsed by the Tomita recognizer first. The resulting parse times are indicated in Figure 4 with a "•". Next, the same strings minus a randomly chosen prefix were given to the substring parser.

[1]The measurements were performed on a SUN Sparc station. The programs were written in Lisp. The time used by the lexical scanner has not been taken into account.

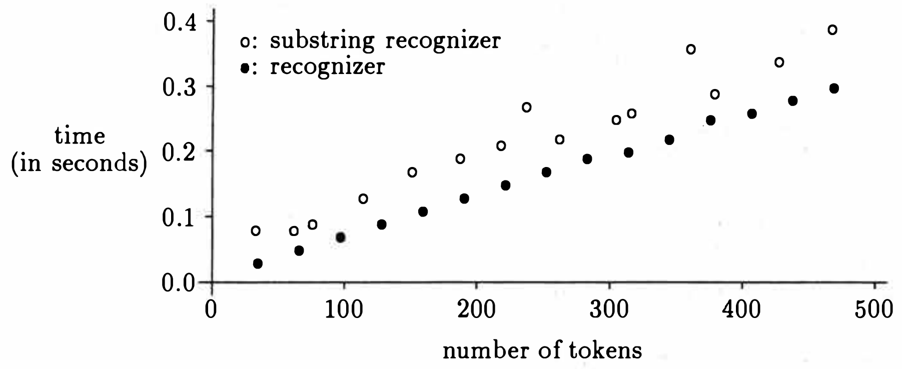

Figure 4: Comparison of the substring recognizer with an ordinary one

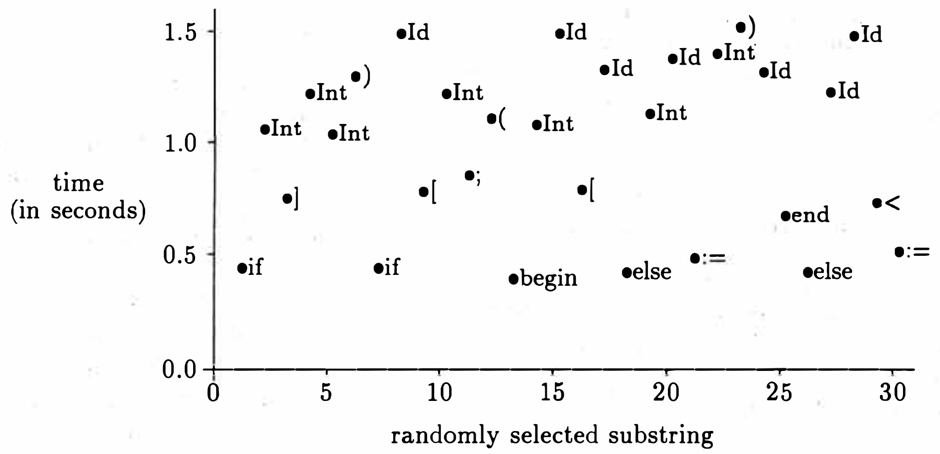

Figure 5: Time needed by the substring parser on Pascal sentences of 100 tokens

The required times are indicated in Figure 4 with a "o".

It turns out that the substring parser has a moderate overhead with respect to the normal parser. This overhead can be interpreted as the time needed for the substring parser to get on the "right track". As Figure 5 shows, the variations in this overhead are caused by the random cutting of the string. For some strings it takes longer than for others to determine of which language construct it can be a substring. The larger the grammar is, the more alternatives are available and therefore the higher the variation.

In Figure 5 we compared the time taken by the substring parser on 30 randomly chosen parts of Pascal sentences of 100 tokens. The dots indicate the amount of time needed and they are attributed with the first symbol of the substring. These measurements show that sentences starting with a token that can appear in many differents contexts, like "Id" or ")", take more time to recognize than sentences starting with a disambiguating token like ":=" or "else".

7 Conclusions

The adaptation of the Tomita algorithm to substring parsing results in a very elegant and powerful algorithm. The main advantage of the fact that it accepts general context-free grammars and uses ordinary LR parse tables is that substring parsing can now be applied in a very general manner, instead of only to carefully written grammars and at the cost of an extra generation phase.

Substring parsing is slower than ordinary parsing, but this will not be a serious drawback for its application as an error recovery technique or as a completion tool. The use of the substring parser in incremental parsing, however, has to be investigated further.

Acknowledgments

We would like to thank Nigel Horspool, who suggested to extend our implementation of the Tomita algorithm to substring parsing. Two years after this discussion we finally saw the need for such a technique and started a serious investigation. Next, we are grateful to Paul Hendriks who pointed out a valuable simplification in the treatment of incomplete reductions in the substring parser, and to Jan Heering for his careful reading of earlier versions of this paper.

References

[1] R. Agrawal and K.D. Detro. An efficient incremental LR parser for grammars with epsilon productions. *Acta Informatica*, 19:369–376, 1983.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] A. Celentano. Incremental LR parsers. *Acta Informatica*, 10:307–321, 1978.

[4] G.V. Cormack. An LR substring parser for noncorrecting syntax error recovery. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 161–169, 1989. Appeared as *SIGPLAN Notices* 24(7).

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] R.W. Floyd. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67, 1964.

[7] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, 1979.

[8] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–579, 1980.

[9] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[10] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 179–191, 1989. Appeared as *SIGPLAN Notices* 24(7).

[11] J.W.C. Koorn. GSE: A generic text and structure editor. Programming Research Group, University of Amsterdam, to appear.

[12] B. Lang. Parsing incomplete sentences. In *Proceedings of the Twelfth International Conference on Computational Linguistics*, pages 365–371, Budapest, 1988. Association for Computational Linguistics.

[13] R. Nozohoor-Farshi. Handling of ill-designed grammars in Tomita's parsing algorithm. In *Proceedings of the International Parsing Workshop '89*, pages 182–192, 1989.

[14] J. Rekers. Parsing for cyclic grammars. Centrum voor Wiskunde en Informatica (CWI), Amsterdam, in preparation.

[15] H. Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, 1985.

[16] G. Snelting. How to build LR parsers which accept incomplete input. *SIGPLAN Notices*, 25(4):51–58, 1990.

[17] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic Publishers, 1985.

[18] D. Yeh. On incremental shift-reduce parsing. *BIT*, 23(1):36–48, 1983.